设计模式

前言

学习设计模式，是为了更好的组织我们的代码，如何解耦，如何真正的达到对修改封闭，对扩展开放的效果。

# 代码质量评价标准

## 可维护性

落地到编码开发，所谓的可维护性，包括修改bug，修改老的代码，添加新的代码之类的工作。

“易维护”指的就是在不破坏原有的代码设计，不引入新bug情况下能够快速的修改或者添加新代码。

这是一个比较主观的概念。

## 可读性

查看代码是否符合编码规范，命名是否能望文生义，注释是否详解，方法是否长短合适，模快划分是否清楚，是否符合高耦合低内聚，

## 可扩展性

在不修改或少量修改原来代码的情况下，通过扩展的方法添加新的功能点代码。

## 灵活性

添加新功能的时候，代码仅需要修改一点点或者已经预先留好了扩展点，或者实现一个功能时原有代码中一经抽象出底层实现代码，或者使用某组接口，这组接口能够应对各种场景。

易复用，易扩展或者易用都可以叫做代码比较灵活。

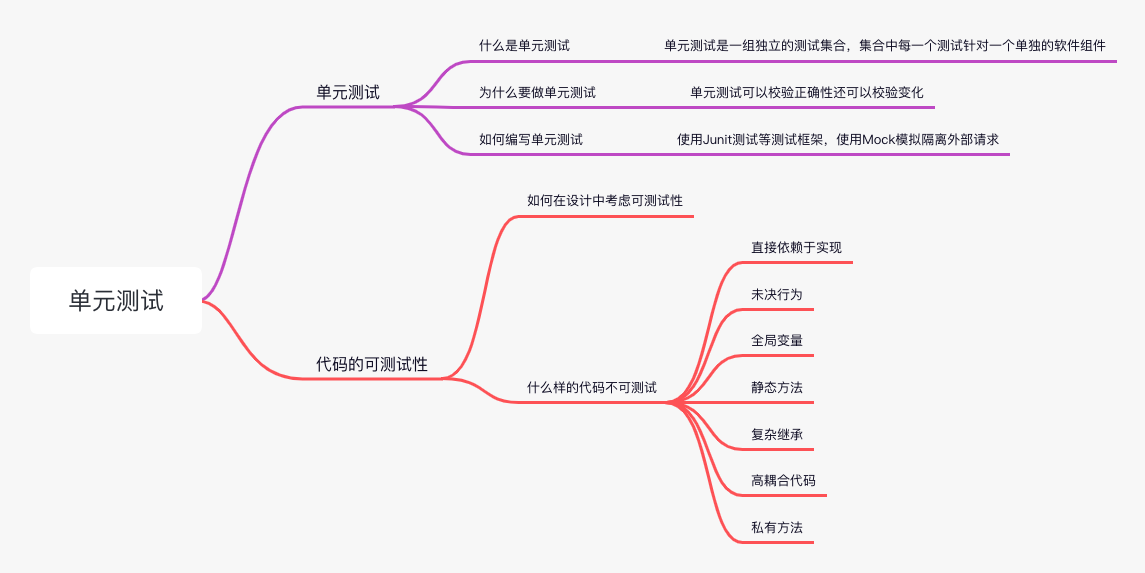
## 简洁性

写代码的时候尽量保持代码简单，逻辑清晰。

## 可复用性

尽量减少重复代码

## 可测试性



* 直接依赖于实现：代码中的组件等直接依赖于其实现，不能替换或者依赖注入。
* 未决行为：未决行为 就是，代码的输出是随机或者说不确定的，比如，跟时间、随机数有关的代码，例如代码中有一个逻辑是根据每个月计算每个月的工作日。如果直接使用系统时间，那么这个函数返回结果就无法测试。
* 全局变量：全局变量是一种面向过程的编程风格。全局变量在多线程中处理复杂，同时也不具备可测试性。单元测试应该彼此独立，隔离，不应该依赖于执行顺序。单元测试的执行不应该有先后顺序，甚至应该允许并发同时执行所有单元测试进而更快速。但是如果出现代码中依赖全局变量的先后执行顺序的逻辑，可能会因单元测试的执行顺序不固定而导致单元测试失败。
* 静态方法：静态方法本身不容易 mock ，因为静态方法本身并非面向对象的而是面向过程的，本身跟全局变量类似。我们编写的静态方法不该依赖于外部环境，系统时间，网络等。
* 复杂继承：设计原则中有个原则是 优先使用组合而非继承。相比组合关系，继承关系的代码结构更加耦合、不灵活，更加不易扩展、不易维护、不易测试。
* 高耦合代码：如果一个类职责不明确，不满足单一职责原则，需要依赖一大堆外部组件才能完成工作，这样的代码耦合度非常高，那我们在编写单元测试的时候，可能需要 mock 一堆依赖的对象。编写单元测试很费劲，几乎无法编写。
* 私有方法：私有方法无法被外部的类方法，也无法从其他类中调用该类的私有方法。无法直接被测试，实际上如果出现这种情况，应该考虑的是这个方法是否应该是私有方法，是否是设计的不合理，是否应该将这个私有方法单独拿出一个接口一个类实现为公有方法。

# 高质量代码方法



## 面向对象分析、设计、编程

面向对象分析（OOA：Object-Oriented Analysis）、面向对象设计（OOD）、面向对象编程（OOP）是面向对象开发的三个环节。

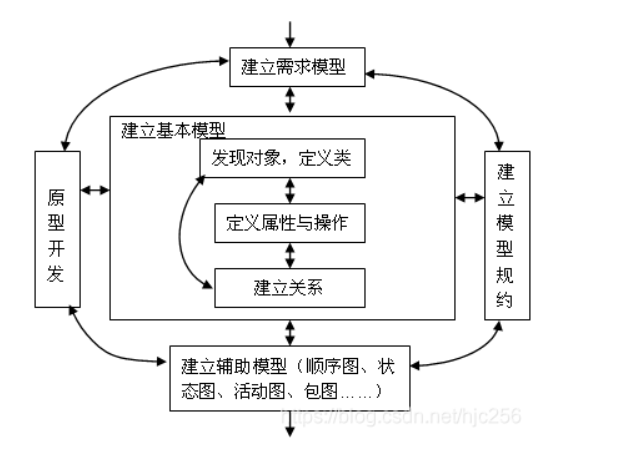
### OOA

OOA模型

* + 需求模型：
    - 用况图：捕获与描述用户的要求
  + 基本模型（类图）：
    - 对象层：给出所有与问题域和系统责任相关的对象，用对象类表示
    - 特征层：定义每个对象类的属性与服务
    - 关系层：通过已定义的关系描述对象类之间的关系
  + 辅助模型
    - * 交互图：完成某项特定功能的一组对象之间的详细交互
      * 状态图：一个对象的状态变迁
      * 活动图：一个服务的流程或业务流程
      * 包图：对关系密切的元素打包，帮助理解系统模型
  + 模型规约：对模型中的所有元素进行详细说明

OOA过程

* + 建立需求模型：定义用况，用用况对用户需求进行规范化描述
  + 建立类图:
    - 发现对象、定义对象类
    - 识别对象的内部特征
    - 识别对象的外部关系
  + 原型开发：结合其他活动反复进行



### OOD

OOA到OOD不是转化，是调整和增补。使OOA作为OOD模型的问题域部分。将OOA阶段分析出来的类定义进行组合继承委托等方式，按照一定的设计原则进行架构设计。

重点是：

* 划分职责进而识别出类
* 定义类及其属性和方法
* 定义类和类之间的交互关系
* 将类组装起来并提供接口执行

## 设计原则

### 单一原则

Single Responsibility Principle,SRP：一个类或者模块只负责一个职责。

判断标准之一就是按照类或模块使用场景进行判断，当前使用背景/场景满足的情况下满足即可。

但是如果类的代码行数，方法，属性过多的时候就需要考虑这个类是否需要重构了。不需要设计大而全的类。尽量做到低耦合，高内聚。

### 开闭原则

Open Closed Principle。对扩展开放，对修改关闭。

添加一个新功能应该是在已有的代码基础上进行扩展(增加模块，类，方法)，而非去对已有的代码进行修改。

实际应用开发中，需要注意的点是：只要修改代码没有破坏原有代码的正常运行，没有破坏原有的单元测试，就可以当做是符合开闭原则的修改。

设计的时候需要时刻保持扩展意识，抽象意识，封装意识。